

James Bulpin

Design and Simulation of a Super-Scalar CPU

Computer Science Tripos Part II

King's College

1999

Proforma

Name: James Bulpin
College: King's College
Examination: Computer Science Tripos Part II, 1999
Word Count: 11,800 (approx.)
Project Originator: Ian Pratt
Project Supervisor: Ian Pratt

Original Aims of the Project

To create two Verilog models of the core of the ARM CPU, one non-superscalar and one superscalar. To compare the versions and look at different configurations for the superscalar version by making the model modular. To synthesis the design and consider critical paths.

Work Completed

Models of both versions were constructed and compared, different configurations of branch prediction and execution units were tried. The aims were quite ambitious and the synthesis of the models was chosen to be dropped in favour of producing a good superscalar model.

Special Difficulties

None.

I James Bulpin of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1	Introduction	1
2	Preparation	3
2.1	Research	3
2.2	Requirements and Limitations	3
2.3	Planning	4
2.4	Verilog	5
2.5	Tools	6
2.5.1	Pre-processor	6
2.6	Design	7
2.6.1	Non Superscalar Version	8
2.6.2	Superscalar Version	10
2.6.3	Design Changes	12
3	Implementation	15
3.1	Non Super-Scalar Model	15
3.1.1	Stage 1: Instruction Fetch (IF)	15
3.1.2	Stage 2: Instruction Decode (ID)	16
3.1.3	Stage 3: Execute (EX)	17
3.1.4	Stage 4: Memory Access (MA)	17
3.1.5	Stage 5: Register Write-Back (WB)	17
3.1.6	The ALU	18

3.1.7	The Register File	18
3.1.8	Program and Data Memory	19
3.1.9	System Integration	20
3.2	Super-Scalar Version	20
3.2.1	Feeder Unit	20
3.2.2	Simple Branch Prediction	21
3.2.3	Dynamic Branch Prediction	21
3.2.4	Instruction Holding	22
3.2.5	Registers	25
3.2.6	Register Colouring	26
3.2.7	Execution Unit(s)	29
3.2.8	Data Memory Access	29
3.2.9	Instruction Scheduling	30
3.2.10	Special Instructions	31
3.2.11	Retirement	31
4	Evaluation	33
4.1	Testing	33
4.1.1	Module Testing	33
4.1.2	Assembler	34
4.1.3	Running Code Fragments	34
4.1.4	Interpreting Output	35
4.2	Comparing the Versions	35

4.2.1	Factorial Program	36
4.2.2	Infrequent Branching Test	37
4.2.3	Benchmarks	38
4.3	Success of the Project	39
5	Conclusions	41
A	Example Module Test Output: datamem.cv	45
A.1	Test Harness	45
A.2	Test Output	46
B	Sample Code: retire.cv	49
C	Some Example Test Programs	55
C.1	Factorial	55
C.2	Infrequent Branching Test	55

List of Figures

2.1	The Method of Working	4
2.2	Non-superscalar Block Diagram	10
2.3	Superscalar Block Diagram	11
2.4	Superscalar Module Diagram	12
3.1	Cell of Register File - Alternatives	19
3.2	Cell of Register File VLSI Layout	19
3.3	Instruction holding ‘slot’	23
3.4	Example architectural to physical register mappings	28
4.1	Pipeline activity table	36
4.2	Number of cycles taken to execute the factorial test	37
4.3	Number of cycles taken to execute the infrequent branch test	38
A.1	Data Memory module test trace output	46

Chapter 1

Introduction

Modern silicon fabrication techniques allow increasingly more transistors on a chip. There is scope for utilising the available transistors to provide more efficient and powerful processors. The key to using this extra capacity is to increase parallelism. If several tasks can be carried out concurrently then the overall throughput should be higher. This is the basis for “pipelining”. The basic principle of a pipeline is to have a number of stages connected together in a sequence. Each acts on an instruction in some way, they all work concurrently on different instructions. A simple pipeline may have three stages: the first fetching and decoding an instruction, the second executing the instruction and the third writing a result back to the registers. At each clock cycle, instructions ‘flow’ down the pipeline for the next phase of processing. It therefore takes three cycles for an instruction to be completely executed but three instructions can be processed at any one time, giving an execution rate of one instruction per second. This approach turns out to be faster than non-pipelined processors because less is done per clock cycle (in a given stage) which means that the clock can run faster.

One problem associated with pipelined designs is when branching, the decision on whether the branch will be taken will be made typically at the execution stage. By the time the decision is made, one instruction (more if the pipeline is longer) will have started its journey through the pipeline. If the branch is taken, this instruction would be on the wrong path of the branch.

A further problem is that it may be necessary to stall the pipeline when waiting for data to be loaded from memory. If the requested data is cached it will take a cycle to retrieve and if the data is not in the cache it may take a while to be fetched from main memory. The pipeline stage responsible for loading must wait which means that the flow of instructions through the pipeline will have to be temporarily blocked (or *stalled*).

Dynamic execution superscalar processors try to overcome these problems and increase the over all instruction rate by executing instructions out of order and predicting the outcome of a branch, “speculating”. Superscalar processors have multiple pipelines in parallel. This obviously adds a great deal of complexity but should give a greater average instruction rate. An important aspect of dynamic execution proces-

sors is their ability to keep execution units busy by finding independent instructions to execute. The branch prediction attempts to guess whether a branch will be taken at an early stage and checking the prediction later. Obviously should an incorrect decision be made, the cost will be high but a lot can be gained by making a good prediction.

Many modern processors are superscalar including the Pentium and Alpha 21164 which do not execute out of order but do execute more than one instruction at a time, and the Alpha 21264 and Pentium Pro which can issue instructions out of order for more efficiency gain.

There currently exists no superscalar implementation of the Advanced RISC Machines Ltd. processor, the "ARM". Current ARMs are relatively simple, they are used as embedded CPUs in devices such as mobile telephones as well as in personal computers such as the Acorn Archimedes.

The aim of this project is to investigate how superscalar techniques might be applied to the ARM instruction set.

Chapter 2

Preparation

2.1 Research

The initial part of the work was to learn about processor microarchitecture, particularly pipelining and superscalar design. This involved revision of work from the IB course on computer design[10] and looking ahead to the Part II architecture course[13]. Other research included looking at the relevant sections of books on the subject[7, 6, 12] and manufacturers' web-sites[1, 16].

It was necessary to know about how the ARM is programmed and an idea of how it works internally. I used the ARM online documentation[1], in particular the ARM 7 specification[8].

2.2 Requirements and Limitations

The basic project requirement was to produce models of the ARM CPU in a hardware description language such as *Verilog* to enable simulation. One model should be a similar architecture to the ARM 7 core, the other should be a superscalar version. The models should produce consistent results given the same ARM-code program.

The detailed requirements and functional specification for the behaviour of the models is essentially the ARM instruction set architecture (ISA)[8]. This defines the behaviour of the CPU as visible to the programmer. A number of restrictions were made in order to make the work feasible in the time available and to concentrate on the core of the CPU rather than peripheral functions. These included eliminating the memory management unit, interrupt handling, the barrel shifter and block data transfers. In order to simplify the design further, the program counter (PC) was made a stand-alone register rather than R15 in the register file, and the link register was not supported.

2.3 Planning

The intention was to use well tried software engineering methods such as the waterfall model (as described by Sommerville[14] and others). A second (conflicting) intention was to work to get observable results as early as possible, and to add functionality to this. The approach used was to design the system first then develop the basic modules with minimum functionality, test each module individually, then integrate them and test the result. Following successful completion of this, more functionality would be added to the modules and they would be tested and reintegrated (and so on). A diagram describing the process is shown in figure 2.1. This is similar in approach to the “rapid prototyping” methods being used in industry.

At every round, a certain function would be added. For example, the non-superscalar version started out only being able to execute ALU instructions with no branching and no hardware interlocking. The next round added hardware interlocking, but no register forwarding. The test programs were written to reflect the limitations, and to test all the functionality added to date, with an emphasis on the newly added function.

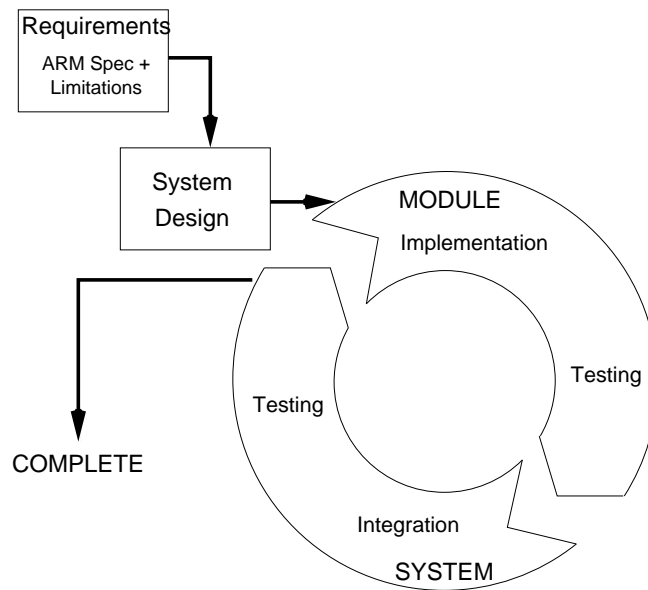


Figure 2.1: The Method of Working

2.4 Verilog

Before coding I had to learn more about the hardware description language (HDL) Verilog, in which the project was written. This involved revision of the IB work with the language and use of tutorials and guides available on the world wide web.

This section is intended to give the reader a brief overview of the capabilities of the language. The lecture notes from the IB ECAD course[4] provide a tutorial on a subset of the language, more in-depth coverage of the language and its syntax can be found in books such as Thomas and Moorby[15].

At a basic level, Verilog (like all HDLs) provides a way to describe a digital electronic circuit using a programming language. Verilog can be used in this way to describe the connections between gates, flip-flops and so on, but what is more useful is its ability to support higher level descriptions of the circuit (or parts of it). For example an 8 bit adder can be coded as either a collection of gates or in a one line statement such as:

```
wire [7:0] result = arg1 + arg2;
```

A very useful feature is register transfer level (RTL) description. This syntax describes what happens to latches at clock edges. For example:

```
always @(posedge clock) countregister <= countregister + 1;
```

Verilog uses modules which are sections of circuit with (possibly) input and output ports (wires). The programmer writes the module, and this can then be instantiated as many times as required in other modules. This approach enables the code to be broken into self contained sections, making the coding more structured and allowing individual modules to be tested before inclusion in other higher level modules (as with procedures in a normal programming language).

A software *simulator* is used to see how the circuit would behave. Simulations can be carried out at various levels, from RTL down to gate level, transistor level or even analogue (for anything except RTL, a *synthesiser* would be used to produce circuit descriptions from the RTL). For this project, the model was simulated at the register transfer level.

2.5 Tools

The Computer Lab Linux machines were initially used to develop and simulate the models. Due to limitations in the otherwise excellent *csim*, it was necessary to use the more powerful *Cadence Verilog XL* on a Sun machine. The centrally managed and backed up filespace was used, accessible from all machines. As well as the regular backups performed by the Lab, periodic copies of the files were taken and backed up on the *Pelican* system.

The version control system *RCS* was used to enable previous versions of the code to be retained without having to keep many different copies of the files. *RCS* is also useful since it helps to prevent accidental deletion of files by keeping the masters in a separate directory and provides a log to track the changes made. *RCS* was also used for this document.

During the project a small number of shell scripts were written to save time. For example, to run a simulation, a script contained all the necessary commands including the appropriate simulator for the architecture. *Emacs* was used to edit the Verilog code, aided by a Verilog “mode” for Emacs.

2.5.1 Pre-processor

Whilst writing the superscalar version of the CPU, it became apparent that a large amount of repetition of code would be needed with only small changes between each line. For example, a list of 64 registers which all need to be updated in a single clock cycle would need lines such as:

```
reg[0] <= (enable)?bus0:32'd0;
...
reg[63] <= (enable)?bus63:32'd0;
```

The only thing changing between them being the number indexing `reg` and the bus number. Since later changes to this were likely, it was preferable to only have to write the line once and have it duplicated for each number 0 to 63.

A major problem I experienced with Verilog is its inability to support a 2 dimensional array of wires (an array of buses). It can handle a 2 dimensional array of registers but not wires. This became a problem when a part of the design needed 64 buses of, for example, 32 bit width, any of which could be assigned to a common bus. This needed many lines of wire declaration and a complicated multiplexer.

The solution was to write a small preprocessor in Perl. This provided duplication facilities as well as basic `DEFINE` support (similar to the normal preprocessor). To duplicate a line, a preprocessor directive (coded as a comment) was given in a style similar to standard `for`-loops:

```
//JRB:DUPLICATE (XX, 0, 63)
reg[XX] <= (enable)?busXX:32'd0;
//JRB:END
```

The effect is the code between the directives was repeated 64 times (in the above example), with any occurrence of `XX` in the code replaced with the current value (somewhere between 0 and 63).

Other directives were:

```
//JRB:DEFINE (AAA, 'BBB')
//JRB:IFDEF(CCC)
...
//JRB:ENDIF
```

The first of these replaced any occurrence of `<AAA>` with `BBB`. The second removed the enclosed code if `CCC` was not defined. This was useful for removing module test wrappers when they were no longer needed.

The preprocessor was run on each source file before simulation using the simulation shell script mentioned in section 2.5. Each source file (`*.cv`) had a corresponding simulation file (`*.sim.cv`), the preprocessor making the latter from the former.

The advantages of using the preprocessor were the saving of time and errors when changing a repeated line and the ability to collect important definitions together (single point of definition).

2.6 Design

Throughout the design and implementation the aim was to make the design as modular as possible, to enable modules to be reused in other parts of the project or for future work. A particular advantage in using modules is being able to swap modules for a particular task. For example, different scheduling modules were experimented with in

the superscalar version. Because they had well defined interfaces, a more sophisticated scheduler could be easily “dropped in” in place of an older one.

With this in mind, it was important to break the entire system down into modules before attempting to code anything. Ideally the interfaces between modules would be defined initially, and these would not change. In reality, as extra functionality is added and implementation details change, the interfaces evolve. Therefore the interfaces defined at the design stage were only guidelines specifying the sort of signals between modules, and their purposes. Some particular signals were named but much of the detail was left until implementation.

Both the first (non-superscalar) design and the later superscalar version have a number of parts in common:

- Arithmetic-logic unit (ALU)
- Memory access and instruction decode stages
- Data memory model
- Other miscellaneous modules.

The decision was taken to code for the non-superscalar version and to modify the appropriate modules for use with the superscalar design later.

2.6.1 Non Superscalar Version

The earlier ARMs use a 3 stage pipeline, the later ones have 5 stages. The latter was chosen for this project as it is more flexible and offers potential for greater performance. The pipeline structure gives a clean module separation for the 5 stages. The design is based on the ‘classic RISC pipeline’ described in the Part II Comparative Architectures course[13] and by Heuring and Jordan[6] amongst others.

Each stage consists of some logic followed by a bank of latches (except the fifth stage which has no latches since its outputs ‘flow through’ directly back into earlier pipeline stages). This gives a basic module design consisting of inputs from the previous stage, some internal logic, some output latches and outputs to the next stage. There are a number of other connections which will be discussed later.

A stage needs to know if the data it is receiving from the previous stage is valid. There are two cases that data coming out from a stage is not valid:

1. The previous stage is still processing and the output data is not ready. For example, whilst waiting for a memory fetch to be completed when the data being output would be garbage until the correct value is ready.
2. The previous stage had no data to process, or terminated the instruction's journey through the pipeline (because of a branch for example).

An extra output from each stage is needed to flag the validity of the data. This signal, to be called `ignore`, is high if the data is invalid. A stage can mark output data as invalid (i.e. should be ignored) in either of the above cases. Alternatively a stage not outputting valid data could push a no-operation (NOP) down the pipe.

There needs to be communication back up the pipeline. There are two reasons for wanting to communicate with earlier stages:

1. If a branch is taken, instructions earlier in the pipeline will be those following the branch. These should not be executed and must be ignored. The execution stage is the only stage that would ever do this. This signal, `flush`, causes the first two stages to mark their output data as invalid so it is not used by further stages.
2. If a stage is still processing its current instruction, it must tell the previous stage to not change its output latches yet as it is not ready and still requires the previous stage's output from the earlier instruction (necessary since the latching is at the end of the previous stage). This signal is called `stall`, it can be sent by any stage to its predecessor, a stage that receives a stall must pass it on to the previous stage.

The execution stage contains an ALU. The ALU was implemented as a separate module to enable reuse in the superscalar design later. The ALU is connected only to this stage so the ALU module can be nested inside the execution stage module.

The registers are all held in a structure known as the register file. This provides a single write port and a number of read ports. The register file module connects to the instruction decode stage (for argument reading) and to the register write-back stage (for writing).

The system memory will be modelled as two separate memories, one for program code and one for data. This reflects the separate instruction and data caches (Harvard architecture) found in many modern processors, including the StrongARM. As core CPU performance is of more interest than memory performance, the models for both memories will be simple. To improve the model for data memory access, all accesses

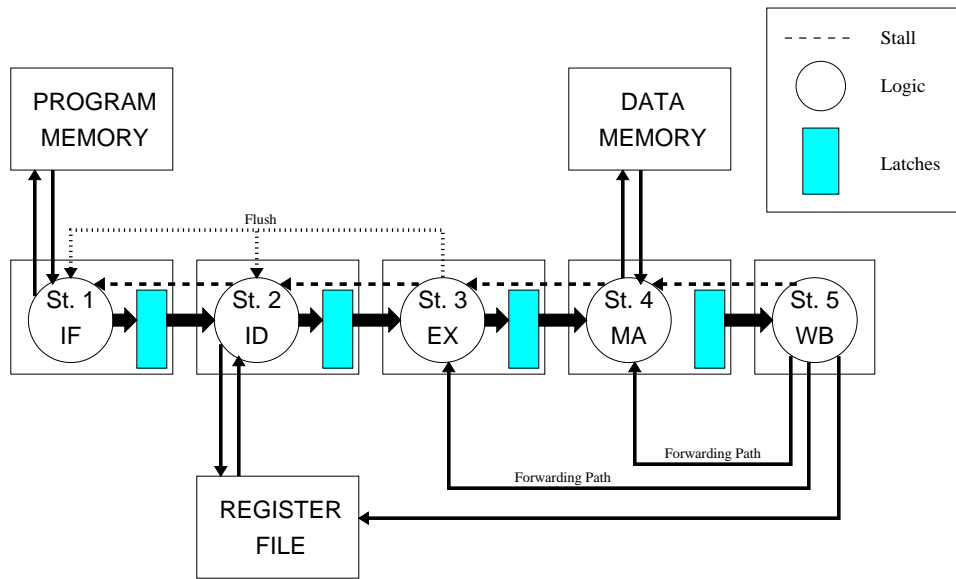


Figure 2.2: Non-superscalar Block Diagram

should go through a module which will provide some kind of variable delay to simulate (e.g.) cache misses. The memories themselves will be modules so could easily be replaced by a more sophisticated design in the future.

The basic block design of the non-superscalar version is shown in figure 2.2.

2.6.2 Superscalar Version

Superscalar machines have multiple pipelines in parallel. There are two key types:

1. Static scheduled: instructions are dispatched to pipelines as they are fetched. It is often hard to keep all the pipelines busy since dependencies between instructions cannot be determined.
2. Dynamically scheduled: the processor looks ahead in the code to determine the dependencies between instructions. It can then decide which ones to dispatch in order to keep the pipelines (execution units) busy. The instructions can therefore be executed out of order, *dynamic execution*.

This design uses dynamic scheduling. This is the more challenging approach but should provide greater gains than static scheduling.

Since branches pose problems when looking ahead in the code, the outcome of a branch will be predicted before it is actually executed. This technique is called *speculation*. The actual outcome is checked later and corrected if necessary.

The superscalar model needs to analyse the code to look for instruction dependencies. The approach used is to rename architectural registers as they are seen with ‘unique’ identifiers, using a new identifier each time a register is written to. This prevents reuse of a register from being viewed as a dependency between instructions which have no direct dependency. This process is called *register colouring*, it is discussed in section 3.2.6.

Once the dependencies of instructions on the results of other instructions is known, it is possible to execute instructions out of order to keep the execution units busy. The only constraint on the order of execution is that for an instruction to be executed, all of its operands must be ready. This means that all previous instructions it depends on have been executed (i.e. causal order is preserved even though total order is not).

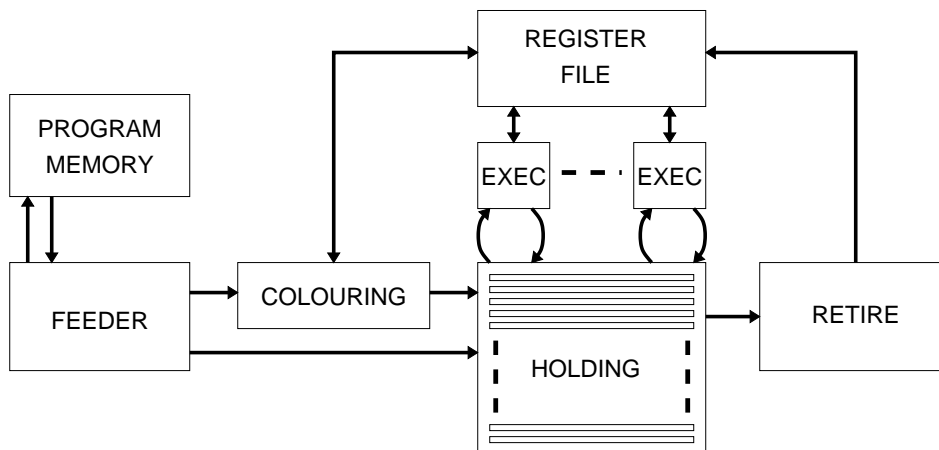


Figure 2.3: Superscalar Block Diagram

The basic design is shown in figure 2.3. The feeder unit provides a stream of decoded instructions. The register colouring unit provides a mapping between physical and architectural registers. The holding unit contains a pool of instructions waiting to be executed or retired, this provides the *window* to ‘look ahead’ in the code. The execution units are controlled by the scheduler, they take a pooled instruction (with both operands ready), waiting to be executed, and execute it, marking it as such. The retirement unit goes through executed instructions in their original order removing them from the pool, updating the architectural state and checking the predicted branch.

Several modules from the non-superscalar version can be reused, some needing modifications. The first two pipeline stages from the earlier version should be com-

bined to make the feeder unit. Major differences between the behaviours of the two versions leading to modifications are:

- The superscalar version needs branch prediction.
- The superscalar version does not need to look up actual register values, only pass the register number out to the register colouring unit.

The ALU and data memory modules can be used with little or no modification.

The module break down is shown in figure 2.4. Some modules are nested or combined because of the large number of connections to other parts. For example, the execution units and register file are in one module with nested ALUs. This is due to the multiplexers required to connect the execution units to the registers to enable simultaneous writing of more than one register.

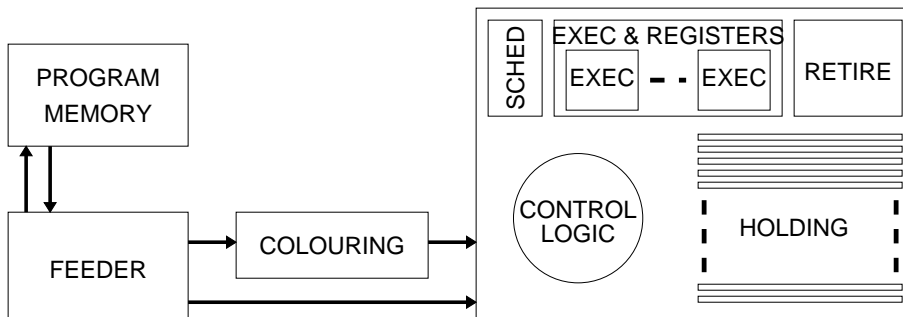


Figure 2.4: Superscalar Module Diagram

2.6.3 Design Changes

As the design progressed a small number of issues came to light that required changes to the limitations of the model. The main issue was the ARM’s ability to predicate execution of any instruction. Every single instruction carries a four bit predicate code (called *condition code* by ARM). The codes represent conditions such as “always”, “carry set” and “zero clear” for example. The instruction is only executed if the status flags (carry, zero, negative and overflow) are in a configuration acceptable by the predicate code of the instruction.

A related feature is the ‘S’ bit in a data processing (e.g. ALU) instruction. When set, this bit causes the status flags to be updated to reflect the result of the instruction. If S is not set, the status flags are unchanged.

This results in extra dependencies between instructions which, after much consideration, proved to be difficult to deal with. It not only enforces order of execution but also creates problems with the register colouring. For example, suppose an instruction `ADDEQ R1, R1, #1` failed to execute because the zero flag was not set, future instructions should use the physical register allocated for the old version of R1 rather than the one for the new version, but by the time the instruction was executed, the registers of incoming instructions would have been coloured for the new version of R1.

The added complexity led to predication of non-branch instructions being unsupported and its implementation left for future work.

Chapter 3

Implementation

This chapter describes how the models were implemented within the block designs described in the previous chapter. The general approach taken was to code the lowest level modules and test each one with a simulation wrapper before moving on to code higher level modules. These were then tested with simulation wrappers. Once all the key modules were completed in this way, the system was integrated and tested. The basic model was then extended to provide extra functionality, one function at a time. This method allowed a working, albeit incomplete, model to be available early in the process, enabling versions with different features to be tested and compared.

3.1 Non Super-Scalar Model

The design of the non-superscalar model was outlined in section 2.6.1. The following sections describe the behaviour of each module.

3.1.1 Stage 1: Instruction Fetch (IF)

The task of the IF stage is to retrieve a word from memory. The program counter (PC) is maintained within the IF stage and addresses the connected program memory.

The stage receives two signals from the second stage, `flush` and `stall`. The former instructs the stage to stop and ignore what it is currently doing, the latter requests a pause for the current cycle. When a `flush` signal is received the `ignore` value is set, on the next clock edge this is loaded into the output latches (along with any data, now garbage) to inform the next stage to ignore the data. The `stall` signal causes the output latches to remain as they were for the next cycle, i.e., the result of the fetch should not be loaded into the latch yet, instead the PC must not be advanced so the input from memory remains the same.

The execute stage is responsible for calculating branches and will send signals to the IF stage to change the PC if a branch is taken. If the `muxselect` line is high, the

offset provided by the execution stage is added to the current PC and 8 subtracted. The subtraction is to take into account the fact that the PC will have advanced from the address at which the branch instruction was loaded by 2 words (8 bytes). Otherwise the PC is incremented by 1 word (4 bytes) to the address of the next instruction.

3.1.2 Stage 2: Instruction Decode (ID)

The ID stage takes the fetched instruction and decodes it to produce a series of values to be used later. The ARM instruction format is regular and most fields can be obtained by taking a subrange of the instruction bits. The different types of instruction have different fields. ALU operations contain the opcode, destination register, and up to two operand registers as well as some other flags. Branch instructions have an offset and a few flags. All instructions have the predicate code bits.

The ‘type’ of the instruction is taken from bits 27 to 25 of the instruction. The bit patterns supported in this model are: 00X - ALU operation (with bit 25 determining whether operand 2 is immediate or from a register), 101 - Branch, 01X - Memory Access. A special pattern of 111 is used in the later superscalar version to denote special instructions used to aid simulation (halt and register dump). The value of ‘type’ is used throughout the design to determine whether certain fields are valid. For example, if the type is 101 then the opcode field will not be valid as it will contain 4 bits of data from a branch instruction.

The decode stage also performs register lookups for operands that are not due to be written by instructions later in the pipeline. A scoreboard in the register file keeps track of whether registers are ‘out for writing’ or not. If a register needed for an operand is not out for writing, its value is retrieved from the register file and passed to the output latches. If the register is ‘out for writing’, flags are set to indicate this and the register number is written on the output latches to be used later when reading the required register value from a forwarding path. Forwarding paths are the mechanism by which register values which are results of instructions in later stages of the pipeline are returned directly to the execution stage if they are needed rather than being written back in the final stage and read from the register file in stage 2. Without this facility, the pipeline would have to stall until the required value was written back.

The ID stage receives signals from its successor stage to request a stall, or to flush the stage (as the IF stage). This stage can generate its own stall signal to send to the IF stage in the circumstance that the destination register is out for writing (i.e. one of the two preceding instructions has the same destination as this instruction, a ‘write after write’ hazard). The stall out is therefore the disjunction of this internally generated stall and the stall in from stage 3.

3.1.3 Stage 3: Execute (EX)

The execute stage is where ALU operations are performed. It contains a nested ALU module (see section 3.1.6). It is at this point that forwarding paths may need to be used. The flags set by the ID stage for forwarding control select which (if any) forwarding path to use. The multiplexers feed the ALU with the opcode controlling which operation is performed. The output from the ALU is passed to the output latches.

The predicate code and the status flags are used to determine if the instruction should be executed. If it should not be, an ignore bit is set to be loaded into the output latches on the next clock edge. A further bit is set to mark the instruction as being valid but not executed so that the target register it has checked out for writing can be checked back in later (but without saving any value).

If the instruction is executed and the S bit set, the status flags (held within the execution unit) are updated to reflect the result of the operation.

If the instruction is a branch instruction and the predicate evaluation resulted in execution, the offset and its corresponding enable line are passed back back to the IF stage to change the PC. The flush signal is sent to the previous stages to instruct them to stop processing the instructions they hold.

3.1.4 Stage 4: Memory Access (MA)

This stage is responsible for memory loads and stores; other instructions pass through unchanged but undergo a one cycle delay as they pass through. ALU instructions cannot simply skip past this stage as this could lead to two instructions both writing to the register file in the same cycle which is not possible with the current register file, and there would be a great deal of complexity with the use of forwarding paths.

The model for data memory is described in section 3.2.8. Stage 4 provides control logic to invoke a read or write is necessary and wait for completion.

3.1.5 Stage 5: Register Write-Back (WB)

The WB stage is the simplest stage, it *checks in* the destination register (if valid) and writes back the value to the register file (if necessary).

3.1.6 The ALU

The ALU performs operations on two 32 bit inputs producing a 32 bit output and 4 status flags. The operations are described in the ARM Specification[8]. In this model, the ALU is implemented as a multiplexer with the 16 functions of the two inputs as the multiplexer inputs with the opcode as the control for the multiplexer. The actual implementation used by Advanced RISC Machines is a series of bit cells connected by a carry look-ahead chain[3]. The ARM implementation is much more compact and efficient but the chosen model is sufficient for this work and should optimise well when synthesised.

3.1.7 The Register File

For this simplified design, at any one time up to two registers may need to be read and one written.

In Verilog, there are two possible ways to write a register value:

1. The register `select` bus could be decoded to give a wire per register, this is high when that register is to be written. This is AND'ed with the write enable signal before clocking the latch that hold this register's value. (Figure 3.1 (1).)
2. All latches are clocked from the master system clock and they each have a multiplexer on their input to choose between the incoming register value and the existing value of the latch. The same decoded, enabled wires from above are used to control the multiplexers, if the control is low, the latch value is fed back to its input thereby maintaining its state, if the control is high, the incoming register value is loaded instead. (Figure 3.1 (2).)

The first of these requires less silicon area and has a shorter total gate delay but the second has the advantage of working on a common clock rather than the delayed gated clock. In a real system the latter is useful because it helps to prevent problems associated with clock skew and allows the circuit to be easily implemented in programmable gate array devices which tend to have special clock distribution wiring (hence needing a common clock).

A read port is easily coded as a multiplexer:

```
portA = registerarray[selectA];
```

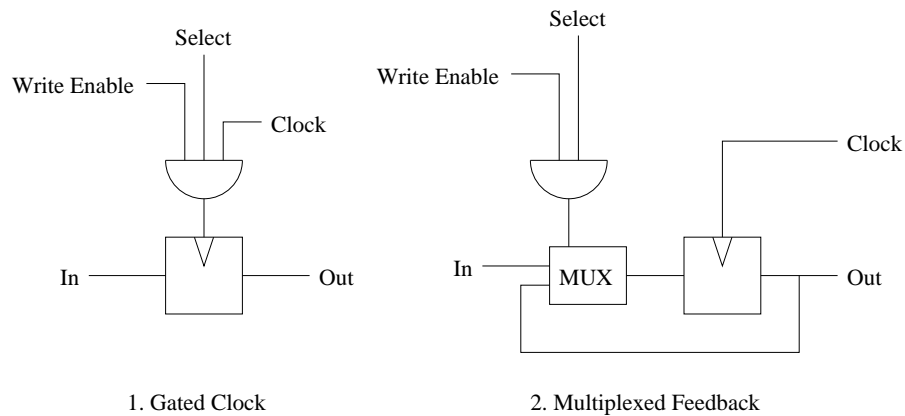


Figure 3.1: Cell of Register File - Alternatives

where the registers and wires have been declared before. This means that any number of read ports is easy to implement in an RTL model but in silicon the size of the structure can get large. In a real register file, latches are unlikely to be used, Furber[3] describes the circuit with which ARM implements the register file. Figure 3.2 shows a possible VLSI layout for a single cell of a 2 read port, 1 write port device, an array of 32 by 16 of these would be needed for a 32 bit 16 register file.

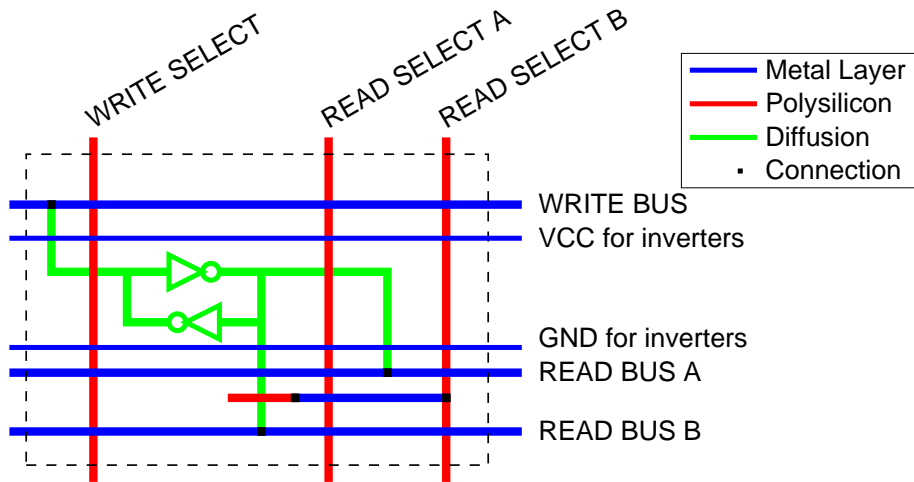


Figure 3.2: Cell of Register File VLSI Layout

3.1.8 Program and Data Memory

The aim of the project is to look at the CPU core so detailed models of the memory are not required. The memory was split into separate areas for program and data, this

is obviously not true of real processors (with exceptions such as the 8051 family) but many modern processors have separate instruction and data first-level caches (Harvard architecture), so this simplification is a valid one to make.

The program memory is derived from the assembler code. The assembler outputs a Verilog file containing a 32 bit wide multiplexer with hard wired inputs corresponding to the instruction code, the particular input is selected by the address bus containing the requested address. This is a purely combinational circuit with no concern for details of cache misses and so on.

The data memory needs to be a little more realistic since the delays associated with memory loads affect the core CPU performance. The model used is described in section 3.2.8.

3.1.9 System Integration

The modules were connected together as shown in figure 2.2. The entire assembled pipeline was connected to a clock generator for simulation. The testing details and results are described in chapter 4.

3.2 Super-Scalar Version

Having implemented the non-superscalar version, the next task was to produce a superscalar CPU. The block design for implementation is shown in figure 2.4 and discussed in section 2.6.2.

3.2.1 Feeder Unit

The feeder unit is the equivalent of stages 1 and 2 from the non-superscalar design, in fact, the same code was used as a starting point (but modified). The main difference between the feeding of this superscalar version and the earlier version is that branch prediction is performed. The idea is that different prediction methods can be tried and encapsulated in modules that have a common interface enabling easy changing of methods. The interface consists of inputs to the predictor module, these are of two types (as well as the clock):

1. Current Status: The PC value at the moment, a flag to denote whether the current instruction is a branch and the branch offset contained within the branch instruction.
2. Feedback: The retirement unit considers branch instructions as it retires instructions in program order. It checks whether the branch should be taken. It passes this decision, along with the branch instruction's address, back to the predictor to enable prediction caches to be updated.

The output from the predictor is the next value of the PC (this will be the old PC plus 4 for non-branch instructions), the 'alternative PC address', that is, the address that should have been used if the prediction turns out to be false and a bit to mark whether the branch was predicted to have been taken or not.

The decoded outputs are essentially the same as the non-superscalar version, the register numbers are those of architectural registers, these are converted to physical registers by the colouring unit later. A major difference from the non-superscalar version is that no register values are looked up at this point.

3.2.2 Simple Branch Prediction

The first branch prediction used was a simple, fixed prediction of either always predicting the branch taken or always predicting the branch not taken. This was initially hard coded into the feeder unit to enable other parts of the design to be tested. Later, this functionality was moved to a separate model. The modules simply take the inputs (branch target address and next location) and pass them to the next PC and alternative outputs (either directly or crossed over depending on which prediction) and keep the prediction flag at a constant value.

3.2.3 Dynamic Branch Prediction

The same interface is used for the more sophisticated prediction modules that were tried later. The 2-bit version implements a *2 bit bi-modal predictor*. The module incorporates 1024 2 bit registers. The address of a register corresponds to the lower 10 bits (actually bits 11 to 2 since bits 1 and 0 will always be zero) of the address of the branch instruction to be predicted. The prediction of the current branch is based on the value of the cache: if the value (can be between 0 and 3) is 2 or 3, the branch is predicted taken.

The 1-bit version uses 1024 1-bit caches. It operates in a similar way to the 2-bit version but the values can only be 0 or 1, denoting not taken and taken respectively.

When (at retire time) a prediction is checked, the address of the branch instruction, its predicted flags and a bit to denote whether the branch was actually taken or not is returned to this module. If a branch was taken, the register addressed by the lower 10 bits of the branch instruction's address is incremented (unless it is already 3 in which case it remains at 3), in a similar manner, if the branch was not taken, the counter is decremented, saturating at 0.

The effect is that previous behaviour is used as a guide for future predictions, one-off changes in behaviour do not have an effect if the counter is saturated at either 0 or 3. The initial value is set to 2, therefore biased towards branches being taken. Analysis of code shows that branches are more often taken than not[13].

The 2-bit predictor ought to perform better than the 1-bit predictor when looping because it is resilient to one-off changes of behavior. For example, if a loop is executed a number of times, both predictors will incorrectly predict the final branch (not taken) but the 1-bit predictor will then incorrectly predict the branch not taken on the next execution whereas the 2-bit predictor will contain the value 2 in its cache so will still (correctly) predict the next branch taken.

The 1024 registers cover a code area of 4kB, there is always the chance that in code currently being executed there could be two branch instructions with the same lower 12 bits (i.e. an exact multiple of 4kB apart) which would cause potentially poor performance of the prediction but pathological cases like these are rare.

3.2.4 Instruction Holding

The *holding unit* contains the pool of instructions waiting to be executed or retired. The pool consists of a number of *slots*, each corresponds to an instruction. A slot is essentially a bank of registers holding the instruction details (decoded), output details (after execution) and some status bits. A block diagram of a slot is shown in figure 3.3.

The slot module encapsulates all the per-slot logic and registers within a single module that can be instantiated as many times as necessary. This is clearly advantageous over a single 2-dimensional array of registers as it breaks the work down into more manageable blocks which can be tested individually.

A slot can be in one of four states: empty, waiting, ready or completed. The empty state is when the slot is not currently holding an instruction and is available for use by an incoming instruction. A waiting slot has been loaded with an instruction but cannot

be executed yet because one or more of its operands are not available. A slot is ready when all of its operands are available and is a candidate for execution whenever the scheduling unit chooses. Once an instruction has been executed, its slot is marked as completed indicating to the retirement unit that it is ready for removal.

The state could be represented by only two wires but for simplicity and ease of understanding, more were chosen:

- **empty** - high if the slot is empty
- **ready** - high if the slot's instruction is ready to be executed
- **complete** - high if the instruction is ready to be retired

To simplify the scheduling, a separate ready line was used for different types of instruction. The three types of instruction were ALU, memory (load/store or 'LS') and branch. Branch instructions do not need executing, they are only required by the retirement unit to check the branch prediction. This leaves two types, ALU and LS with corresponding ready flags **ready** and **readyLS** respectively.

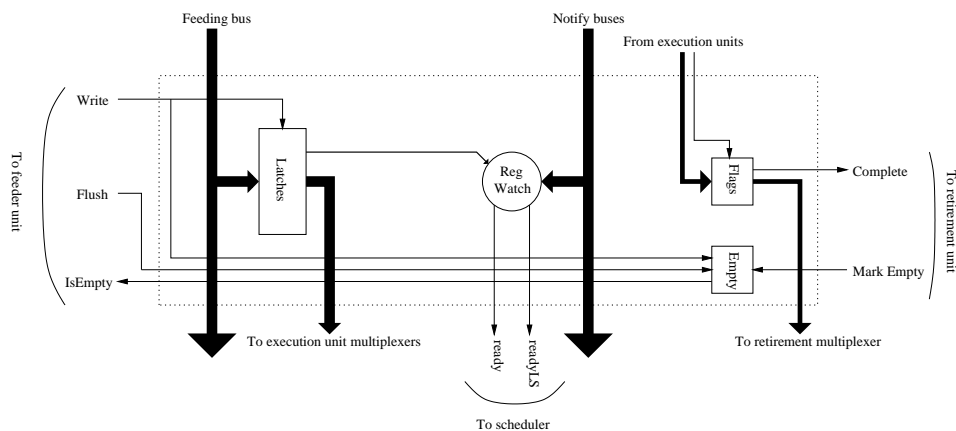


Figure 3.3: Instruction holding 'slot'

The logic internal to the slot updates these flags by watching signals coming in on buses to which the slots are connected. The principle activities are:

- **Loading**

An instruction is initially loaded into a slot after it arrives from the feeder unit. The slots are arranged in a ring. A counter keeps track of the next slot in the ring, if it is empty, an incoming instruction is inserted in the slot and the counter advanced. This ensures that the instructions are in program order in the slots so

that when they are retired using a similar counter, they are retired in order as well.

When the slot sees a high `write` line, it loads the incoming instruction into registers and marks its `empty` flag low to indicate it is full. It also checks if any of the instruction's operands are about to become ready.

- **Watching for registers**

An instruction will be loaded with the numbers of the physical registers (P-regs, see section 3.2.5 for register details) it needs (along with enable flags to say if it actually needs the registers). The slot could store the actual value of the register once it has been computed or alternatively the values could be stored centrally and a flag set to indicate it is ready. The latter method was chosen since it requires far less wiring and allows all the P-regs to be held together close to the execution units, providing performance benefits in a real implementation. It also simplifies the case when a register is computed before an instruction needing it has been loaded so will need to be stored somewhere in the interim, the chosen method allows this with little extra overhead.

When a register value is about to be ready (i.e. available on the next cycle), the register number is sent out on a notification bus (with an enable signal). The slots listen to these buses, and upon spotting the a register number needed by the instruction, a flag marking this register as available is set in the slot. Since each execution unit can have a register ready at the same time, there needs to be a notification bus and enable for each execution unit and the slots must listen to each.

The `ready` and `readyLS` flags are combinatorial functions of the 'register available' flags denoting whether registers are needed or not (e.g. a move instruction does not need operand 1 and an ALU instruction with an immediate value does not need operand 2).

- **Execution**

The slot makes all its instruction information available on output buses. These are connected to multiplexers controlled by the schedulers. A slot will not know when its instruction is being executed so the execution units will signal back to it when it has been completed. This is simply a completion enable line that when high causes the slot to set its `complete` flag.

- **Retirement**

Similarly all details required by the retire unit are always available on the outputs so the retire unit must send a signal to instruct the slot to mark itself as empty, ready to be reused later.

In the initial implementation, the unit was sized to contain 64 slots. The hold unit also contains the scheduler and execution units as nested modules. This arrangement makes the wiring less complex since there are many interconnections between slots, the scheduler and the execution units. The hold unit handles the incoming decoded instruction stream. It maintains the slot counter mentioned earlier and provides multiplexers for the execution and retirement units. These multiplexers are controlled by the scheduler and retirement unit respectively.

The decoded instruction comes from the feeder unit (via register colouring) in a way similar to way the decode stage feeds the execute stage in the non-superscalar design. If the next slot is not empty the hold unit must stall the feeder until the slot becomes free.

3.2.5 Registers

Both physical and architectural register banks are maintained. The former holds the actual values of the coloured registers, the latter holds architectural state over execution restarts following an incorrect branch prediction. Since the execution units make heavy use of both types of registers, they are located in the module containing the execution units.

The register file logic provides the following functions:

1. Multiplexors for the inputs to execution units.

The execution control circuitry receives addresses of the register(s) needed and flags to indicate whether the value(s) should come from an architectural or physical register (see section 3.2.6). The register unit provides a multiplexer controlled by these signals to provide the actual value for the execution unit.

2. Simultaneous write-back from execution units.

There are a total of three execution units (see section 3.2.7), all of which could complete in the same cycle. This causes up to three values to be written back simultaneously to the register file. This is achieved by having a multiplexer on each register's input selecting between the execution units. (There will never be two instructions with the same destination register in the pool at any one time so this will work.)

3. Transfer of a value from a physical to architectural register.

Retirement (section 3.2.11) causes values to be copied from a physical to an architectural register. This is the only time an architectural register is written to, so they only need a single write port. Since this design only allows one

retirement per cycle, a single multiplexer between the physical register outputs and the single architectural register input controlled by the retirement unit is sufficient.

There are three execution units taking two operands each and one copy-to-architectural multiplexer, this means that the physical register file needs seven read ports.

3.2.6 Register Colouring

The purpose of the register colouring unit is to translate between architectural and physical registers and vice versa. It maintains three lookup tables:

1. Architectural to physical (used when feeding)
2. Physical to Architectural (used by retirement unit)
3. Free Physical registers list

The process of register colouring removes the *name dependence* between instructions leaving only *data dependence*, undoing the live variable analysis and register colouring performed by the compiler[11]. The idea of a *live range* is used, a live range for a register begins when it is written, and ends when that value is last read. A new physical register is used for each live range.

The mechanisms used to update the tables are:

- Beginning a Live Range

When a destination register that will be written to (i.e. only valid instructions which write a result register) is encountered, the ‘forward’ lookup table (architectural to physical) is updated for the architectural destination register with the next free physical register from the free list. The ‘reverse’ lookup table is updated by inserting the architectural register in the location for the physical register used. The free list is updated to mark this physical register as ‘currently in use’.

- A Physical Register becoming free

Once a live range ends, a physical register must be marked as free so that it may be reused. A live range will be declared to have ended when the architectural register is written to next. This means that when *retiring* an instruction

that writes to a register, the physical register corresponding to the architectural register in the range *just ending* can be marked as free. This can be done since instructions after the one being retired will refer to the physical register corresponding to the architectural register in the live range *just beginning*. The only instructions referring to the physical register being marked as free are ones that are before the currently retiring one, which would already have been retired.

To perform this action, it is necessary to know the last physical register to be allocated to the architectural register found by a reverse lookup on the physical destination register of the retiring instruction. This is implemented with a further look up table called ‘`last`’ which is indexed by physical register number, and returns the number of the physical register last allocated to the same architectural register. When allocating a new physical register, a forward lookup is performed on the architectural register before the new pairing is written. This value is then put in the `last` table in the location for the new physical register. If this is the first time this architectural register has been written to since the last flush (or beginning of the code) then the last value will be meaningless since there is no live range to mark as free. The physical register value 0 was used as a marker for this (an alternative way would be to have a corresponding bitmap indicating validity).

Lookups are far simpler due to the two table operation. Given an architectural register (for an operand), the physical register can be found simply by using a multiplexer on the output of the lookup table: `physical = forwardlookup[architectural]`. The reverse look up is similar.

When beginning a new code block (i.e. after a flush or from the very start), the values in the lookup tables will be either empty, or will contain mappings created after the wrongly predicted branch so will be meaningless. The correct values of the architectural registers are held in an architectural register bank and are updated at retire time. Using the method discussed so far, lookups for operands will produce physical registers containing incorrect values. What is needed is to get the values from the architectural register file until the physical registers are written to again.

There are a number of solutions to this problem:

1. The lookup tables could be preloaded with mappings such as architectural register `x` maps to physical register `x` and all 16 architectural register values copied to corresponding physical registers.
2. Each architectural register could be tagged to indicate whether the value to use should be obtained from the architectural or physical register banks. This tag

would be reset to mark the architectural bank and switched to mark the physical bank as soon as that architectural register is written to.

3. Registers could be copied from the architectural to physical registers on demand. The normal mechanisms for finding a free register would be used. This would need tags to record whether a copy is needed but the instructions would not need to carry any more information.

The second of these methods was used since it saves a great deal of data copying which takes time and adds complexity to the circuit and requires extra read and write ports on the register file. When looking up an operand, if the tag for that architectural register is set then a bit in the decoded output is set to tell the execution unit to use the architectural register instead of physical register. The architectural register number is stored in the lower four bits of the register number field.

Figure 3.4 shows an example of the forward and reverse lookup tables after some instructions have been fed through the register colouring unit.

Forward		Reverse	
Archi	Phys	Phys	Archi
0	23	0	0
1	4	1	4
2	5	2	4
3	45	3	10
4	2	4	1
5	54	5	2
6	8	6	11
7	31	7	12
8	49	8	6
9	27	9	13
10	13	10	6
11	6	11	5
12	44	12	10
13	9	13	9
14	43	14	4
15	22	15	11
		16	3
		17	0
		18	0
	

Figure 3.4: Example architectural to physical register mappings

3.2.7 Execution Unit(s)

Different execution units are needed for ALU instructions and memory access instructions. The execution units are given the operand register numbers or immediate data with flags to specify which is to be used as well as instruction data including the operation code (opcode) for the ALU.

The registers are needed by all the execution units so could be held in a central register file module with ports for each execution unit. The actual method used was to have the registers in a single execution unit central module. This performs all the register lookups and decisions between register values and immediate data. Nested within this are separate modules for each execution unit which are passed the actual values. The execution unit modules return completion information back to the central execution module which in turn provides notifications back to the slots for registers being ready and instructions being completed. Status flags (carry etc.) values are written back to the slots and left to the retirement unit to use if necessary.

The ALU and parts of the memory access code were reused from the non-superscalar design.

3.2.8 Data Memory Access

In order to make simulations realistic, models of the data memory are needed. An accurate model is beyond the scope of this project. Instead one giving the impression of quasi-random access delays is used. The fine details of caching algorithms and memory behaviour will not be modelled. If future work required a better module, one could be written using the same interface and would work with the rest of the code unchanged.

This module has a constantly running counter, incremented on each clock pulse. The counter counts from 0 to 7 and round again. A data value is deemed to be ready when this counter reaches zero. For example, if a value is requested and the counter is at 3, the flag to tell the execution unit that the data is ready is not set until 5 cycles later when the counter reaches 0. Therefore two consecutive requests will be served 8 cycles apart. This is not realistic, since even if the requests are for the same address, there is a delay. The memory itself is modeled as 1024 32 bit registers (i.e. 4kB). Writing is immediate, this is quite close to behaviour observed in systems with write buffers or write-back first-level data caches.

3.2.9 Instruction Scheduling

The task of the instruction scheduler is to choose an instruction that is ready to be executed and dispatch it to the correct type of execution unit. The `ready` and `readyLS` signals form two buses of 64 lines which feed into a scheduler module, the outputs of this module are the addresses of the slots for each execution unit, each with an enable line. These outputs control the multiplexers between the slots and the execution units.

There are a number of possible scheduling algorithms but to get best performance, it is necessary to minimise the time the retirement unit spends waiting for the next slot to complete. This means that slots should have completed before the retirement unit gets to them. The scheduler can help achieve this by scheduling the oldest instructions first.

The scheduler itself is a separate module. This enables different designs to be tried with ease. The module needs to schedule for both ALU and load/store instructions. The latter is easier since, in the current design, there is only one load/store execution unit, but there are two ALU execution units so care must be taken to ensure that the same instruction is not sent to both at the same time. The scheduler for the load/store unit is separate (although in the same module) from the ALU scheduler. The load/store scheduler takes the 64 bit array of `readyLS` flags as input and produces the 6 bit address as output, using a simple priority encoder to select the first high bit in the array. The ALU scheduler works in a similar way, the part of the module scheduling for the first ALU unit uses a priority encoder as with the load/store scheduler, the part scheduling for the second ALU unit uses a modified priority encoder that excludes the slot number scheduled by the first ALU scheduler.

This simple scheduler implementation suffers from one drawback, when the feeding wraps around and starts filling up the first slot again and these first slots become ready, the priority encoder will choose them over ones which come later in the holding unit but are actually older. There are two possibilities here:

1. Live with it. In a number of cases the instructions recently fed to the first slots of the hold unit will depend on results of the slots towards the end of the unit so will not be marked as ready until the slots from the far end of the unit are executed. There will also be cases where the data dependencies are weaker and ready instructions are left at the end of the unit, blocking retirement, until all the earlier ones have been executed.
2. Use the retirement unit current slot counter to hint to the scheduler where the earlier instructions are in the unit. The module interface makes a provision for this (the counter wires are brought to the interface) but the current implemen-

tation does not use the facility. This method is likely to use a great deal more logic with corresponding longer signal delays but should provide a much better scheduling algorithm which would cope with all cases. An example is using a barrel rotator on the 64 input lines before they enter the priority encoder. The degree of rotation would be controlled by the current slot counter such as to bring the line corresponding to the oldest instruction to the top.

A compromise would be to use the top few bits of the retirement counter as a hint to the scheduler as to where it should start looking. This should need less logic than a fully optimised scheduler but would not suffer from the problems described in (1) above.

3.2.10 Special Instructions

To enable debugging and testing some extra instructions were provided. HLT stops execution when it is encountered *at retire time*. The purpose of this is to stop the simulation once all the code has been executed. Code past the HLT instruction will have started its journey through the system but will not be retired and hence not change the architectural state. REG causes the architectural state to be dumped by the simulation when this instruction is retired. The actions of these instructions takes place at retirement since that is the one area that architectural state is correct. The instructions are handled in the same way as NOP instructions (another addition for testing purposes), they are marked as complete when being fed and cause no change to architectural or physical state.

3.2.11 Retirement

Retirement is the process of removing completed instructions from the pool and committing changes to architecturally visible state. Instructions are retired in program order. A counter pointing into the pool is maintained. Each time an instruction is retired, the counter is advanced to point to the next instruction, which is retired when ready.

The retirement unit checks to see if the instruction has a destination register value to write. If so, it uses the register colouring unit's lookup tables to find which architectural register to use. It then sends signals to the register file to copy the value from the physical to architectural register. This ensures that at any point the architectural registers contain the correct values for the point in the program that the retirement unit has reached.

The retirement unit maintains a set of architectural status flags. It uses the flags from the instruction (obtained during execution) and the *S* bit to update the flags if necessary. Because instructions are retired in program order, the value of the status flags is correct for the current point in the program.

When a branch is encountered, its predicate bits and the architectural status flags are used to work out if the branch should have been taken. This is compared with the marker bit held with the instruction that came from the branch predictor. If the prediction was correct, a signal is sent back to the predictor to confirm and retirement carries on. If the prediction failed, the predictor is informed and the alternative address is sent back to the feeder to get the PC to jump to the correct address. At the same time, the entire holding pool, colouring lookups and feeder are flushed since they contain instructions past the wrongly predicted branch. Because the architectural registers and status flags have been updated in program order, they represent the correct state at the point the branch occurs. Thus, when execution resumes from the correct location, the architectural state will be correct.

Chapter 4

Evaluation

4.1 Testing

Throughout the work, the approach taken was to test as often as possible. This took the form of individual tests for modules and tests on the integrated system after new functions were added.

4.1.1 Module Testing

Many of the modules coded had a module test wrapper written for them (the exceptions being those that really did not need it or proved difficult to test in isolation). The test harness, a module called `SIMSYS()`, was included in the same Verilog source file as the module to be tested. The `SIMSYS` module was enclosed in an `IFDEF(testing)/ENDIF` block so that it was only passed to the simulator if `testing` had been defined previously.

This method allowed the behaviour of each module to be looked at in isolation. The `SIMSYS` module generally contained a clock source and some input stimuli that could easily be changed. The effect was similar to being able to inject a signal into a part of a circuit without worrying about the other parts. Both simulator traces and textual output from the `$display` commands were used to view the behaviour. An example of the output of a module test is shown in appendix A, along with the test harness used to generate it.

Each module was tested with a number of different input stimuli (stimuli changed by editing the test harness code). Boundary conditions and likely trouble spots were tested as well as general cases. Test results were compared with what would be expected. The comparison was done by hand. A more sophisticated method would use automated testing in which a set of test vectors (detailing input stimuli and expected output) would be presented to the test suite (similar to how real hardware is tested).

4.1.2 Assembler

From the proposal stage through to the design and implementation stages it was known that a number of restrictions on the ARM instruction set architecture (ISA) would have to be made in order to make the project feasible in the time available. This meant that arbitrary ARM code could not always be run because it may not conform to these restrictions. I chose to write my own minimal assembler with which I could precisely control which features were allowed.

A second reason for wanting my own assembler was for the ability to add extra commands. I added 3 control commands not normally available: `NOP`, `HLT` and `REG` which stand for ‘no-operation’, ‘halt’ and ‘dump register file’ respectively. These were for simulation purposes. This proved easier than modifying the GNU assembler since I know Perl well but do not know C or the details of the GNU assembler.

The assembler was a small Perl program. It read an assembler file into memory and performed two passes over the code, one to establish the addresses corresponding to labels and a second to actually produce the code. It produced two outputs: a Verilog model of the program memory, and a binary file for use with a real ARM or an ARM emulator. The program was written in such a way as to allow extra commands to be added with ease.

4.1.3 Running Code Fragments

As soon as enough modules had been completed to allow them to be integrated, small code fragments were executed, each aimed at testing a particular aspect of the functionality. The assembler described in section 4.1.2 was used to assemble the program memory model, which was then executed.

The very first test was on the non-superscalar version before any interlocking or forwarding paths were added. The code had interleaved `NOP`'s (as in software interlocking) between a series of ALU instructions. At this stage, the branching had not been fully implemented, so only linear sequences were tried. The register values were displayed each cycle by `$display` commands in the code and these were compared to the expected values calculated by hand.

When branching had been fully implemented, simple unconditional branches were tried first. The code had some linear sequences of ALU operations followed by a jump to another address then some more ALU operations. Again the expected results were manually calculated and compared to the behaviour observed, including the correct

flushing of the pipeline. Conditional branches were next, used to create simple loops, a simple multiplication loop was used often as a test:

```

@ Test a simple loop
MOV R1, #20      @number of times to go around loop
MOV R4, #0      @accumulating result
:loop
ADD R4, R4, #8  @add 8 to acc
SUB S R1, R1,#1 @decrement loop counter
B (NE) loop     @branch if loop counter not at zero
MOV R8, R4      @move result to R8

```

A number of code fragments were tried. After each major function was added, the emphasis of the tests was on this new function but other functions were tested to ensure that the addition had not damaged anything else.

4.1.4 Interpreting Output

Virtually all modules had `$display` commands producing output to the screen on each clock edge, some only displayed information when particular flags were high or when state changed but most displayed current state on every cycle. This meant that a large amount of output data was produced when running anything more than a simple module or code fragment test. To make the interpretation of the output easier, a small Perl script was written to turn the output from the non-superscalar simulation into an HTML table describing the state of the pipeline.

An example from one such table is shown in figure 4.1. The first column is time, the next five columns represent the five stages and the final column shows the instruction currently in the execute stage. A blue colour denotes a stage not currently active (the values of the signals are therefore garbage), a red colour indicates that the stage is stalled, awaiting the following stage(s) to complete.

4.2 Comparing the Versions

The different versions were compared by executing the same program on each and counting the number of clock cycles used to complete the program. The following sections describe a couple of the test programs that were used.

Instruction	PC	STALLED	IGNORE	Instruction	cvnz	cond	ALWAYS	IGNORE	branched	arg1	reg	STALLED	cond	failed	arg2	reg	isbranch	opcode	MOV	arg1	arg2	destreg	destreg	isalu	result	dest	STALLED	IGNORE	value	Instruction
450	e08ba00c	0	0	e618b000	0000	ALWAYS	0	0	0	0	0	0	0	0	13	0	0	MOV	0	0	0000002d	8	00000017	00000017	0	0	0	00000017	MOV (ALWAYS) 8	
550	00000000	0	0	e08ba00c	0000	ALWAYS	0	0	0	8	0	0	0	0	0	0	0	AND	8	0	00000000	0	00000000	00000000	0	0	0	00000000	AND (ALWAYS) 11	
650	00000000	1	1	00000000	0000	ALWAYS	0	0	0	0	0	0	0	0	0	0	0								11	0	0	0000002d		

Figure 4.1: Pipeline activity table

4.2.1 Factorial Program

A small factorial program was written. It has a subroutine to multiply two registers and place the result in a third, and a loop to multiply (using this routine) an ‘accumulator’ register by the loop index value which is decremented on each iteration until it reaches 1. The pseudocode for this is (the assembler code is shown in appendix C):

```
integer acc := 1;
integer fact := 5; //factorial to compute
while (fact != 0) do
    integer temp := multiply(fact, acc)
    acc := temp
    fact := fact - 1
endwhile
```

The largest factorial that can be represented as a 32 bit integer (signed or unsigned) is 12! (0x1C8CFC00). The program was run twice on each model, once computing 5! and once computing 12!. On completion, the result was checked against the expected value and the number of cycles taken to run the program recorded. The results are

shown in figure 4.2. The last cycle to be counted is the one where the final result is written to its architectural register.

Model	5!	12!
Non-Superscalar	119	525
SS (2 ALU units) with static 'always branch' predictor	97	374
SS (2 ALU units) with dynamic 1-bit branch predictor	106	396
SS (2 ALU units) with dynamic 2-bit branch predictor	86	366

Figure 4.2: Number of cycles taken to execute the factorial test

These results show a definite gain in instruction throughput in the superscalar version over the non-superscalar version, 43% for 12! with the 2-bit predictor. This is obviously only one particular example which does not represent the range of programs that could be executed but gives a reasonable indication of the success of the superscalar methods.

The results suggest that (for this example at least) the 1-bit dynamic predictor is worse than the static predictor. This can be explained by the fact that this particular program contains branches that are usually taken, so the static predictor only makes an incorrect prediction when the loop is not taken. The 1-bit predictor does what was correct last time, so after incorrectly predicting a branch, the following time that branch is executed, the prediction will be that it is not taken but in this program it always is. Effectively there are twice as many incorrect predictions.

4.2.2 Infrequent Branching Test

A program like the factorial one above is very well suited to superscalar processors because the correct branch prediction can be achieved a high percentage of the time. The program used in the following test was designed to demonstrate that some code suffers a performance loss when superscalar techniques are used.

This program contains a large loop, it is executed 12 times with a register, initially set to 12 being decremented at the end of each loop. The loop contains 12 blocks of instructions, each block contains a test which compares the loop index to the block's own number (e.g. the third block compares the index to 3, fourth to 4, and so on.). If the values compared are equal, the program branches forward to skip past the next instruction, otherwise the instruction is executed (it is only a simple ALU operation). The assembly program is shown in appendix C, the operation can be described as:

```
for i := 12 to 1 step -1 do
```

```

if i != 1 then a[1] = a[1] + i
if i != 2 then a[2] = a[2] + i
...
if i != 12 then a[12] = a[12] + i

```

The loop branch is taken all but the last time, the other branches are only ever taken once and to the predictor, they come at an unpredictable time. This suggests that predicting the branches within the loop could be unsuccessful. Figure 4.3 shows the results of this test. This shows that the superscalar design performs worse than the non-superscalar one. This is because the branch prediction fails so often and there is a large overhead when a prediction fails so the overall time is longer.

Model	Cycles
Non-Superscalar	521
SS with static 'always branch' predictor	more than 800
SS with static 'don't branch' predictor	542
SS with dynamic 1-bit branch predictor	575
SS with dynamic 2-bit branch predictor	551

Figure 4.3: Number of cycles taken to execute the infrequent branch test

A useful observation from this test is that the static 'always branch' predictor performs very badly (because most branches are not taken). It was observed in the factorial test that the performance of this predictor and the 2-bit dynamic predictor were similar but in this case the dynamic version has a huge advantage. This suggests (as would be expected) that the dynamic predictor is the better predictor to use in order to cope with different types of code.

4.2.3 Benchmarks

It would be a useful exercise to run SPEC benchmark tests on the models. SPEC 95 (Standard Performance Evaluation Corporation) benchmarks test various aspects of a processor's performance. The integer suite, CINT95, is a set of 8 integer C programs designed to reflect the real workloads of current processors. The programs are large and test not only core CPU performance but caching, compiler performance and memory speed. This coupled with the fact that the full ARM CPU would need to be implemented means that in order to use them on these models, they would have to be heavily modified so have not been used. A further problem is that the execution times would have been prohibitively long. SPEC 95 benchmarks run for several minutes on real hardware so running them on a software simulation would take too long: an 800

clock cycle test took about 4 seconds, giving a 200Hz clock which is a factor of 1 million slower than real hardware, suggesting that each minute on a real processor would be equivalent to a couple of years on a simulator!

4.3 Success of the Project

The aims of the project were to develop non-superscalar and superscalar Verilog models of the core of a simplified ARM processor, to look at different configurations for the superscalar design and to compare the versions. The two models have been written and code can be run on them.

The modular design has allowed different designs for some modules to be tried, an example being different branch prediction methods. Small programs were run on different versions of the non-superscalar and superscalar designs and the results compared.

Programs have been run on both models with successful results. The programs must conform to certain constraints such as not using block-data instructions or program counter manipulations other than branching. A number of tests were carried out demonstrating the fact the models do work. The tests are not exhaustive but are representative of the functions supported and I am confident that the behaviour is correct.

I am satisfied that the modelling, simulation and comparison aims have been achieved and the work was successful. The project could provide a good base on which to build a full RTL model of the ARM processor.

Chapter 5

Conclusions

The work carried out has demonstrated:

1. Superscalar techniques applied to the ARM can improve instructions-per-cycle efficiency. In the tests carried out, an improvement was noticed in many cases.
2. The complexity of the circuitry needed for a superscalar design is far greater than a traditional pipeline. The line count following pre-processing of the Verilog files comes to about 13,000 for the superscalar version and 3,600 for the non-superscalar version.

This has consequences for both the physical size and power consumption of an ARM CPU. The ARM is designed to be low power and small, so a four-fold increase in both may counteract any gain in speed when choosing a processor for a particular application.

3. The ARM has features that do not work particularly well with a simple superscalar design. For example, the predicated execution of any instruction adds the potential for many more dependencies between instructions, reducing the impact of the superscalar benefit. There are ways to cope with this particular problem such as viewing predicated instructions in a similar way to branches but they would add yet more complexity.

One of the most beneficial gains from the superscalar approach over the pipelined design was the branch prediction. It was (in general) successful in reducing wasted cycles but with only a relatively small amount of extra hardware needed compared to the out of order execution. It may be this fact that inspired the design of the ARM 10[9], which contains a branch prediction unit in the fetch stage of the standard five stage pipeline. The ARM 10 has a limited amount of parallel execution but is not superscalar. The leading architect of the ARM 10 Thumb Family is quoted on the subject:

To keep the area and power down, we avoided the complexity and cost of a full superscalar machine.[2]

With the benefit of hindsight, if doing the project again I would spend more time on defining the interfaces initially and I would try to support awkward features such as predicated execution from an early stage.

Bibliography

- [1] <http://www.armltd.co.uk/documentation/>.
- [2] <http://www.arm.com/coinfo/pressrel/15oct98/index.html>.
- [3] S.B. Furber. *VLSI RISC Architecture and Organization*. Marcel Dekker, Inc., 1989.
- [4] D.J. Greaves and S.W. Moore. *ECAD and Architecture Practical Classes*. University of Cambridge Computer Laboratory, 1997.
- [5] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitative Approach (2nd Edition)*. Morgan Kaufmann Publishers, Inc., 1996.
- [6] V.P. Heuring and H.F. Jordan. *Computer Systems Design and Architecture*. Addison Wesley Longman Inc., 1997.
- [7] M. Johnson. *Superscalar Processor Design*. Prentice Hall, 1991.
- [8] Advanced RISC Machines. *ARM7 Data Sheet (Doc. No.: ARM DDI 0020C)*. Advanced RISC Machines Ltd., 1994.
- [9] Advanced RISC Machines. *ARM 10 Thumb Family Product Overview*. Advanced RISC Machines Ltd., 1998.
- [10] S.W. Moore. *Computer Design*. University of Cambridge Computer Laboratory, 1997.
- [11] A Mycroft. *Optimising Compilation*. University of Cambridge Computer Laboratory, 1998.
- [12] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann Publishers, Inc., 1994.
- [13] I.A. Pratt. *Comparative Architectures*. University of Cambridge Computer Laboratory, 1998.
- [14] I. Sommerville. *Software Engineering (4th Edition)*. Addison Wesley, 1992.
- [15] D.E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer, 1991.
- [16] http://developer.intel.com/drg/mmx/manuals/dg/dg_chp2.htm.

Appendix A

Example Module Test Output: datamem.cv

This is a sample of the output from the module test on the data memory module and the code for the test harness.

A.1 Test Harness

```
//JRB:IFDEF (TESTING)
module SIMSYS ();
  wire clk, enable, rnw, ready, busy;
  wire [11:0] addr;
  wire [31:0] datar, dataw;

  CLK10MHz clock(clk);

  //Module to test
  datamemory dmemA(clk, addr, dataw, datar, enable, rnw, ready, busy);

  //Keep count of cycles so far, will do things on specific cycles
  reg [7:0] counter;
  initial counter <= 8'd0;
  always @(posedge clk) counter <= counter + 1;

  //Do things on certain cycles
  assign addr = (counter == 3)?12'd4:(counter==20)?12'd5:(counter==40)?12'd4:12'd0;
  assign dataw = 32'h123456;
  assign enable = ((counter==3)||((counter==20)||((counter==40)))?1:0;
  assign rnw = ((counter==20)||((counter==40)))?1:0;

  //display current state
  always @(posedge clk) begin
    $display("%t: counter=%d A=%d DW=%h DR=%h en=%b rnw=%b rd=%b by=%b",
             $time, counter, addr, dataw, datar, enable, rnw, ready, busy);
  end

endmodule // SIMSYS
//JRB:ENDIF
```

A.2 Test Output

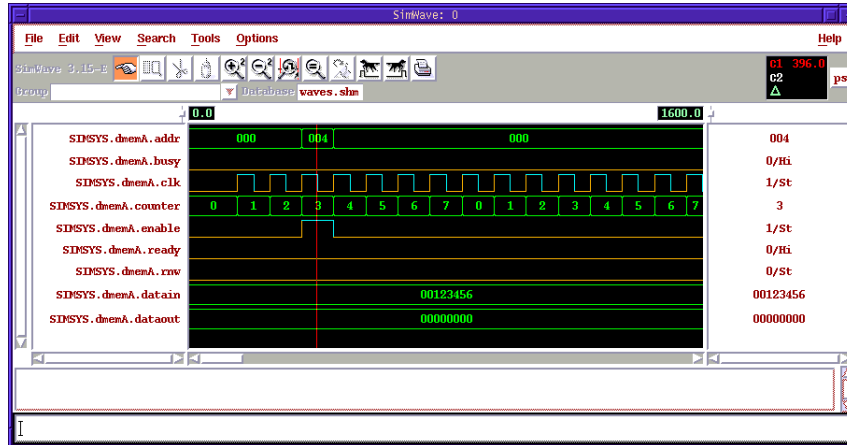


Figure A.1: Data Memory module test trace output

```

150: counter= 0 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
250: counter= 1 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
350: counter= 2 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
450: counter= 3 A= 4 DW=00123456 DR=00000000 en=1 rnw=0 rd=0 by=0
550: counter= 4 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
650: counter= 5 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
750: counter= 6 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
850: counter= 7 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
950: counter= 8 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1050: counter= 9 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1150: counter= 10 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1250: counter= 11 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1350: counter= 12 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1450: counter= 13 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1550: counter= 14 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1650: counter= 15 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1750: counter= 16 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1850: counter= 17 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
1950: counter= 18 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
2050: counter= 19 A= 0 DW=00123456 DR=00000000 en=0 rnw=0 rd=0 by=0
2150: counter= 20 A= 5 DW=00123456 DR=00000000 en=1 rnw=1 rd=0 by=0
2250: counter= 21 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=1
2350: counter= 22 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=1
2450: counter= 23 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=1
2550: counter= 24 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=1
2650: counter= 25 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=1 by=1
2750: counter= 26 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
2850: counter= 27 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
2950: counter= 28 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3050: counter= 29 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3150: counter= 30 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3250: counter= 31 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3350: counter= 32 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3450: counter= 33 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3550: counter= 34 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3650: counter= 35 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0

```

```
3750: counter= 36 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3850: counter= 37 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
3950: counter= 38 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
4050: counter= 39 A= 0 DW=00123456 DR=00000005 en=0 rnw=0 rd=0 by=0
4150: counter= 40 A= 4 DW=00123456 DR=00000005 en=1 rnw=1 rd=0 by=0
4250: counter= 41 A= 0 DW=00123456 DR=00123456 en=0 rnw=0 rd=0 by=1
4350: counter= 42 A= 0 DW=00123456 DR=00123456 en=0 rnw=0 rd=0 by=1
4450: counter= 43 A= 0 DW=00123456 DR=00123456 en=0 rnw=0 rd=0 by=1
4550: counter= 44 A= 0 DW=00123456 DR=00123456 en=0 rnw=0 rd=0 by=1
```


Appendix B

Sample Code: retire.cv

The following is the Verilog code for the retirement unit in the superscalar version of the CPU.

```
// $Id: retire.cv,v 1.13 1999/05/03 14:13:33 jrb44 Exp $
// $Date: 1999/05/03 14:13:33 $

// James Bulpin
// Pt II Project 1998/99
// Design and Simulation of a Super-scalar CPU

// Retirement unit. Reads completed instruction lines from holding bay,
// updates status flags and so on, marks instruction line as empty.

// J RB:DEFINE(TESTING, "1")
// JRB:DEFINE(REGTOP, "5")

module retire (linetoread, readenable, //output to request next line
               type, complete, S, flagsin, //data in from line
               cond, dest, data, instaddr, //data in from line
               markempty, //to line
               clk,
               flushout, altaddr, //branch control
               retplys, reten, //register copy P->A
               dumpArchi, //back to exec unit
               fbPC, fbbranched, fbenable //branch predict feedback
               );

//Outputs to request line
output [5:0] linetoread; //address of the line to read
output readenable; //high if we actually want to read

//To stop and restart from a different address than predicted
output flushout; //to flush rest of CPU
output [31:0] altaddr; //new address for feeder

//Data from line requested
input [2:0] type; //type of instruction
input complete;
input S;
input [3:0] flagsin;
input [3:0] cond; //condition code
input [31:0] data; //type-specific data (alt addr for branch)
input [31:0] instaddr; //original address of instruction
input [<REGTOP>:0] dest; //destination P-reg

//Feed back to line
```

```

output      markempty;    //to mark this line as empty

//Connections to colouring unit to allow P-regs to be reused
output [<REGTOP>:0] retplys;    //phys reg of retired dest
output      reten;        //enable for this

//Branch predict feedback
output [31:0]      fbPC;        //address of branch instruction
output      fbbranched;    //did it branch?
output      fbenable;    //enable for both

//Other connections
input      clk;            //system clock
output     dumpArchi;    //high to tell exec unit to $display A's

//Internal wiring
wire      condfailed;    //high if condition code not satisfied
//Aliases for flags in from line
wire      NF;
wire      ZF;
wire      CF;
wire      VF;
//Aliases for type in instruction being retired
wire      isALU      = !type[2] && !type[1];
wire      isBranch   = type[2] && !type[1] && type[0];
wire      isSpecial  = type[2] && type[1] && type[0];
wire      isLS       = !type[2] && type[1];

//internal state
reg [5:0]      currentline;
reg [3:0]      flags;

//aliases for flags
assign NF = flags[3];
assign ZF = flags[2];
assign CF = flags[1];
assign VF = flags[0];

//calculate whether condition code is satisfied
assign condfailed = (
  (cond == 4'b0000)? !ZF :
  (cond == 4'b0001)? ZF :
  (cond == 4'b0010)? !CF :
  (cond == 4'b0011)? CF :
  (cond == 4'b0100)? !NF :
  (cond == 4'b0101)? NF :
  (cond == 4'b0110)? !VF :
  (cond == 4'b0111)? VF :
  (cond == 4'b1000)? (!CF || ZF) :
  (cond == 4'b1001)? (CF || !ZF) :
  (cond == 4'b1010)? (NF ^ VF) :
  (cond == 4'b1011)? (!NF ^ VF) :
  (cond == 4'b1100)? (ZF || (NF ^ VF)) :
  (cond == 4'b1101)? (!ZF || (!NF ^ VF)) :
  (cond == 4'b1110)? 0 :
  (cond == 4'b1111)? 1 :0);

//Check whether branch prediction was true, using S as guess marker
//1 => predict branch taken
//0 => not taken

```

```

wire  predictfailed = isBranch && (S == condfailed);
assign flushout     = predictfailed && complete;
    //flush if we predicted wrong
//if predicted wrong, fallback address in type-specific data, send
//this back to feeder
assign altaddr = data;

//Branch prediction feedback
assign fbPC        = instaddr;    //original address of instruction
assign fbbranched = !condfailed; //branched if cond success
assign fbenable    = isBranch;    //only enable if actually a branch

//on clock edge, if complete then update local status flags and increment
//counter
//only update stuff if condfailed = 0, still increment though
always @(posedge clk) begin
    flags        <= (readenable && complete && S && !condfailed && isALU)
        ?flagsin:flags;
    currentline <= (predictfailed)?6'd0:(readenable && complete)
        ?currentline+1:currentline;
end

//connect counter to output
assign linetoread = currentline;

//mark this line as empty iff it was complete and read was enabled
assign markempty  = readenable && complete;

//Can't think why we would not read in a given cycle (other than not
//complete) so... TODO check
assign readenable = 1;

//Connect back to colouring unit
assign retpphys = dest;
assign reten    = (isALU || isLS) && complete;

//initialise on startup
initial begin
    currentline <= 6'd0;
    flags        <= 4'd0;
end

//reporting
always @(posedge clk) begin
    $display("%t %m: currentline=%d flags=%b type=%b", $time, currentline,
        flags, type);
    if (predictfailed && complete) $display("%t %m: Predict Failed goto %h",
        $time, altaddr);
    if (!predictfailed && complete && isBranch)
        $display("%t %m: Predicted Correctly!",
            $time);
    if (complete && isBranch)
        $display("%t %m: Predict: S=%b condfailed=%b cond=%b",
            $time, S, condfailed, cond);
end

//Check special instructions
always @(posedge clk) begin
    if (isSpecial && (data[1:0] == 2'd1)) begin

```

```

$display ("STOPPED DUE TO HLT");
$stop; //HLT
    end
    end
    assign dumpArchi = isSpecial && (data[1:0] == 2'd2); //REG

endmodule // retire

//Module test wrapper

//JRB:IFDEF(TESTING)
module SIMSYS();
    wire [5:0] linetoread;
    wire      readenable;
    wire [2:0] type;
    wire      complete, S;
    wire [3:0] flagsin, cond;
    wire      markempty, clk;

    CLK10MHz clock(clk);
    retire retA (linetoread, readenable, //output to request next line
                type, complete, S, flagsin,
                cond, //data in from line
                markempty, //to line
                clk
                );

    reg [10:0] counter;
    always @(posedge clk) counter <= counter + 1;
    initial counter <= 0;

    //pretend every line is complete except 5
    assign complete = (counter == 5)?0:1;

    assign type = 3'd0;
    assign S = 1;
    assign flagsin = 4'd3;
    assign cond = 4'd3;

endmodule
//JRB:ENDIF

/*
 * $Log: retire.cv,v $
 * Revision 1.11 1999/03/30 14:19:23 jrb44
 * bug fixes.
 *
 * Revision 1.10 1999/03/27 16:30:43 jrb44
 * Retires loads/stores properly.
 * BP feedback logic.
 * instaddr connections.
 *
 * Revision 1.9 1999/03/23 14:53:19 jrb44
 * Support for special instructions (HLT...)
 * HLT handling code.
 * REG handling (connect back to exec unit)
 *
 * Revision 1.8 1999/03/22 17:46:46 jrb44
 * Added more prediction reporting.

```

```
* Corrected flag reading behaviour.
*
* Revision 1.7 1999/03/17 23:33:18 jrb44
* Corrected reten function.
*
* Revision 1.6 1999/03/17 14:15:52 jrb44
* Tidied code, added extra reporting.
*
* Revision 1.5 1999/03/17 01:06:01 jrb44
* Branch prediction checking.
* Flush connections
* alt-addr support.
* Dest passing to colouring unit.
*
* Revision 1.4 1999/03/05 12:09:42 jrb44
* Hold testing completed on this phase.
*
* Revision 1.3 1999/03/02 16:00:29 jrb44
* Added module test wrapper. Tested.
*
* Revision 1.2 1999/03/02 13:58:33 jrb44
* Connected counter to output.
*
* Revision 1.1 1999/02/17 19:41:31 jrb44
* Initial revision
*
*/
```


Appendix C

Some Example Test Programs

C.1 Factorial

```
@ $Id$
@ Factorial program
@ Compute factorial of value in R3 (not preserved) R3 must be nonzero
@ Result returned in R4
MOV R3, #12      @00
MOV R4, #1       @05
:loopf
@REG
MOV R0, R3      @08
MOV R1, R4      @0C
B mult          @10
:return
MOV R4, R2      @14
SUB S R3, R3, #1 @18
B (NE) loopf    @1C
NOP             @20
NOP             @24
NOP             @28
REG
HLT             @2C

@SUBROUTINES
:mult
@multiply R0 by R1, result in R2, R0 must be nonzero
MOV R2, #0      @30
NOP
:loop
ADD R2, R2, R1  @34
SUB S R0, R0, #1 @38
B (NE) loop     @3C
B return       @40

@ $Log$
```

C.2 Infrequent Branching Test

```
@ $Id$
@ This test should branch seldomly
```

```

MOV R13, #12 @Counts iterations of outer loop
:outer
SUB S R14, R13, #1
B (EQ) skipi
ADD R1, R1, R13
:skipi
SUB S R14, R13, #2
B (EQ) skipii
ADD R2, R2, R13
:skipii
SUB S R14, R13, #3
B (EQ) skipiii
ADD R3, R3, R13
:skipiii
SUB S R14, R13, #4
B (EQ) skipiv
ADD R4, R4, R13
:skipiv
SUB S R14, R13, #5
B (EQ) skipv
ADD R5, R5, R13
:skipv
SUB S R14, R13, #6
B (EQ) skipvi
ADD R6, R6, R13
:skipvi
SUB S R14, R13, #7
B (EQ) skipvii
ADD R7, R7, R13
:skipvii
SUB S R14, R13, #8
B (EQ) skipviii
ADD R8, R8, R13
:skipviii
SUB S R14, R13, #9
B (EQ) skipix
ADD R9, R9, R13
:skipix
SUB S R14, R13, #10
B (EQ) skipx
ADD R10, R10, R13
:skipx
SUB S R14, R13, #11
B (EQ) skipxi
ADD R11, R11, R13
:skipxi
SUB S R14, R13, #12
B (EQ) skipxii
ADD R12, R12, R13
:skipxii
SUB S R13, R13, #1
NOP
NOP
B (NE) outer
MOV R14, #123

```

@Expected completion results:

@R1 = 77 0x4D

@R2 = 76 0x4C

@R3 = 75 0x4B

@ ...

@R12 = 66 0x42

@R13 = 0 0x00

@R14 = 123 0x7C

@ \$Log\$

Project Proposal

James Bulpin (jrb44)
King's College

Computer Science Tripos Part II Project Proposal

Design and Simulation of a Super-Scalar CPU

October 1998

Project Originator: Ian Pratt

Special Resources: 1. CL Account with 25Mb backed up space.
2. Access to SRG Linux boxes.

Signatures:

Project Supervisor Ian Pratt

Signature:

Director of Studies Dr Ken Moody

Signature:

Overseers Dr Ross Anderson and Dr Frank King

Introduction

Super-scalar processors are those that can execute more than one instruction at a time. The technique used to enable this is 'Dynamic Execution' which covers the processing techniques of:

Branch Prediction: The processor looks ahead in the code and tries to predict if a branch will be taken. This means that the processor need not wait until the branch instruction has been executed before starting any following instructions.

Dataflow Analysis: In order to execute two instructions in parallel, it must be known that the instruction that came later in the code does not depend on the result of the instruction that came earlier as this result would not be ready until both instructions had completed. Dataflow analysis decides which instructions are dependant on each other's results and works out an optimised execution schedule.

Speculative Execution: There will be times when not all of the pipelines will be in use, especially near branches or complicated interdependent instructions. To make the most of the processing capacity, instructions can be executed speculatively, if their results are not needed, they can just be ignored and as that processing power would not have been used anyway, nothing is lost. If however they are needed then the spare processing capacity would have been used to advantage.

There is a considerable amount of literature available on this subject including:

- Computer Architecture, A Quantitative Approach, 2nd Edition, Hennessy and Paterson, 1996
- Superscalar Microprocessor Design, Mike Johnson, Prentice Hall 1991

ARM Ltd also make a great deal of information available along with various tools to emulate the ARM processor.

The project aim is to write a Verilog super-scalar CPU and to experiment on it with different functional unit configurations and dynamic scheduling algorithms.

Required Resources

In order to run the simulations, which require several CPU-hours, I will need access to the System Research Group's Linux machines.

The Verilog code is likely to become quite large especially when generating flattened files. I want to use RCS so I will require about 25Mb on space on a CL account so I can access it from the SRG Linux machines. As this is regularly backed up, I will not have to worry about my own backups.

Starting Point

I am starting the project with knowledge of processor architecture from Simon Moore's IB course on Computer Design and from some background reading when choosing this project. I know a small amount of Verilog, as used in the IB practical classes. The preparation for the project will involve further reading into the detail of processor design and the techniques used in super-scalar projects. This will require access to the material used in Ian Pratt's Part II course in Comparative Architectures, which I have obtained a copy of.

I will also need to extend my knowledge of Verilog as the IB course only covered a small subset of the language.

Project Description

In order to experiment with different structures and algorithms for dynamic execution, it is necessary to have a model that can be used. The aim is to write a Verilog implementation of a CPU and use that as the basis for experimentation. The Verilog CPU can be simulated with a tool such as 'csim'.

Only the core processor will be modelled. The extra functions such as the memory management unit will not be modelled. A simulation wrapper will be written to model the behaviour of the rest of the system.

The instruction set and programming model of a real CPU will be used as the specification of the Verilog CPU as this will enable test programs to be run on the simulated CPU and the results compared against a program run on the real CPU. The internal architecture of the Verilog CPU will not necessarily match that of the real CPU it is based on.

The CPU whose instruction set will be used will be the ARM as it is a RISC processor which is a good base for a super-scalar design and there are many tools and much documentation available for it, primarily though the ARM University Program.

The work breaks down into a number of stages:

Firstly a simple, non super-scalar Verilog implementation will be written. This will allow the design to be tested against the real ARM by executing some test programs which will be written to exercise as many aspects of the CPU's operation as possible. The ARM Toolkit will be used to help construct these programs and the simulator (the ARMulator) will be used to check that the simulated CPU gives the same output as a real ARM CPU would. The average number of instructions per cycle will be recorded.

Once the basic model has been constructed and verified, it can be made super-scalar. This will involve adding a second pipeline and the modules to perform dataflow analysis and branch prediction. This can then be tested as before and the average number of instructions per cycle recorded.

The design will be modular to allow easy changes of architecture. This will allow experiments to be carried out, for example experimenting with different numbers of functional units or different algorithms for the dynamic scheduling.

Up to this point, the Verilog will have been written to model the behaviour of the CPU, with no detail of gate level design. Therefore delay will not have been taken into account. The next phase of the project will be to use a synthesis tool to generate a gate level design which can be used to identify which parts of the design require the highest number of gates and where the critical path is. This should allow the determination of a maximum clock rate.

Extensions

This is a very open ended project and would naturally lead to the following extensions which, dependent on time, may be carried out:

- Target a particular technology to produce a back annotated net list. This will allow a more realistic clock rate to be identified. There is also the possibility of programming the CPU's net list into a FPGA in order to have an actual working (albeit slow) CPU.
- Numerous other experiments with the architecture and gate level layout.
- Implementation of the extra functionality of the ARM not previously modelled (such as the memory management unit).

Plan of Work

The work will be broken down into the following segments:

Preparation:

1. Background reading about processor design and in particular about super-scalar design.
2. Learning about the ARM instruction set and programming model in detail.

Practical Work:

3. Write Verilog for the simple CPU.
4. Test and improve on this model.
5. Write Verilog code for dynamic execution extensions (scheduler, branch predictor...)
6. Test new model and experiment with architecture.
7. Synthesise model and do critical path analysis and try to optimise.

Report:

8. Initial plan of report, draft of each section to decide on content.
9. Write report detail.
10. Final reading over, presentation, binding and so on.

Each of these segments should take about 2 weeks with the exception of 1, 2 4 and 10 which should take about 1 week each, the first starting when confirmation of the project's acceptance has been received. This should mean that the first four segments will take place during Michaelmas Term, 5 during the period between the end of full term and the end of term. 6 and 7 during the first half of Lent Term and the final three during the second half of Lent term and the period between the end of Lent full term and the end of Lent term. I aim to have the entire project finished by the Easter vacation but in the event of difficulty the Easter vacation can be used to finish off any unfinished work.